SIAM J. SCI. COMPUT.
Vol. 38, No. 5, pp. S143–S155

EFFICIENT MESH MANAGEMENT IN FIREDRAKE USING PETSC DMPLEX*

MICHAEL LANGE†, LAWRENCE MITCHELL‡, MATTHEW G. KNEPLEY§, AND GERARD J. GORMAN¶

Abstract. The use of composable abstractions allows the application of new and established algorithms to a wide range of problems, while automatically inheriting the benefits of well-known performance optimizations. This work highlights the composition of the PETSc DMPlex domain topology abstraction with the Firedrake automated finite element system to create a PDE solving environment that combines expressiveness, flexibility, and high performance. We describe how Firedrake utilizes DMPlex to provide the indirection maps required for finite element assembly, while supporting various mesh input formats and runtime domain decomposition. In particular, we describe how DMPlex and its accompanying data structures allow the generic creation of user-defined discretizations, while utilizing data layout optimizations that improve cache coherency and ensure overlapped communication during assembly computation.

Key words. mesh, topology, partitioning, renumbering, Firedrake, PETSc

AMS subject classifications. 65Y05, 65Y15, 65M60, 68N99

DOI. 10.1137/15M1026092

1. Introduction.

The separation of model description from implementation facilitates multilayered software stacks consisting of highly specialized components that allow performance optimization to happen at multiple levels, ranging from global data layout transformations to local kernel optimizations. A key challenge in designing such multilayered systems is the choice of abstractions to employ, where a high degree of specialization needs to be complemented with the generality required to facilitate the utilization of third-party libraries and thus promote code reuse. The use of high-level domain-specific languages (DSLs) and composable abstractions allows existing algorithms and optimizations to be inserted into this hierarchical framework and applied to a much wider range of problems.

In this paper we describe the integration of the DMPlex mesh topology abstraction provided by the PETSc library [2] with Firedrake, a generalized system for the automation of the solution of partial differential equations (PDEs) using the finite element method (FEM) [23]. We outline how DMPlex is utilized in Firedrake to provide

*Received by the editors June 16, 2015; accepted for publication (in revised form) December 21, 2015; published electronically October 27, 2016. This work was supported by the embedded CSE programme of the ARCHER UK National Supercomputing Service (http://www.archer.ac.uk) and the Intel Parallel Computing Center program through grants to both the University of Chicago and Imperial College London.

http://www.siam.org/journals/sisc/38-5/M102609.html

†Corresponding author. Department of Earth Science and Engineering, Imperial College London, London SW7 2AZ, UK (michael.lange@imperial.ac.uk). The work of this author was supported by EPSRC grants EP/M019721/1 and EP/L000407/1.

‡Department of Mathematics and Department of Computing, Imperial College London, London SW7 2AZ, UK (lawrence.mitchell@imperial.ac.uk). The work of this author was supported by NERC grant NE/K006789/1.

§Computational and Applied Mathematics, Rice University, Houston, TX 77005 (knepley@gmail.com). The work of this author was partially supported by DOE contract DE-AC02-06CH11357 and NSF grant OCI-1147680.

¶Department of Earth Science and Engineering, Imperial College London, London SW7 2AZ, UK (g.gorman@imperial.ac.uk). The work of this author was supported by EPSRC grants EP/M019721/1 and EP/L000407/1.

the required mapping between topological entities and degrees of freedom (DoFs), while supporting various mesh input formats, runtime domain decomposition, and mesh renumbering techniques. In particular, we describe how DMPlex and its accompanying data structures allow the generic creation of user-defined discretizations, while utilizing data layout optimizations that optimize cache coherency and ensure computation-communication overlap during finite element assembly.

2. Background.

2.1. Firedrake. Firedrake is a novel tool for the automated solution of finite element problems defined in the Unified Form Language (UFL) [1], a DSL for the specification of PDEs in weak form pioneered by the FEniCS project [19]. Firedrake imposes a clear separation of concerns between the definition of the problem, the local discretization defining the computational kernel used to compute the solution, and the parallel execution of this kernel over a given data set [23]. These multiple layers of abstraction allow various types of optimization to be applied during the solution process, ranging from high-level caching of mathematical forms to compiler-level optimizations that leverage threading and vectorization intrinsics within the assembly kernels.

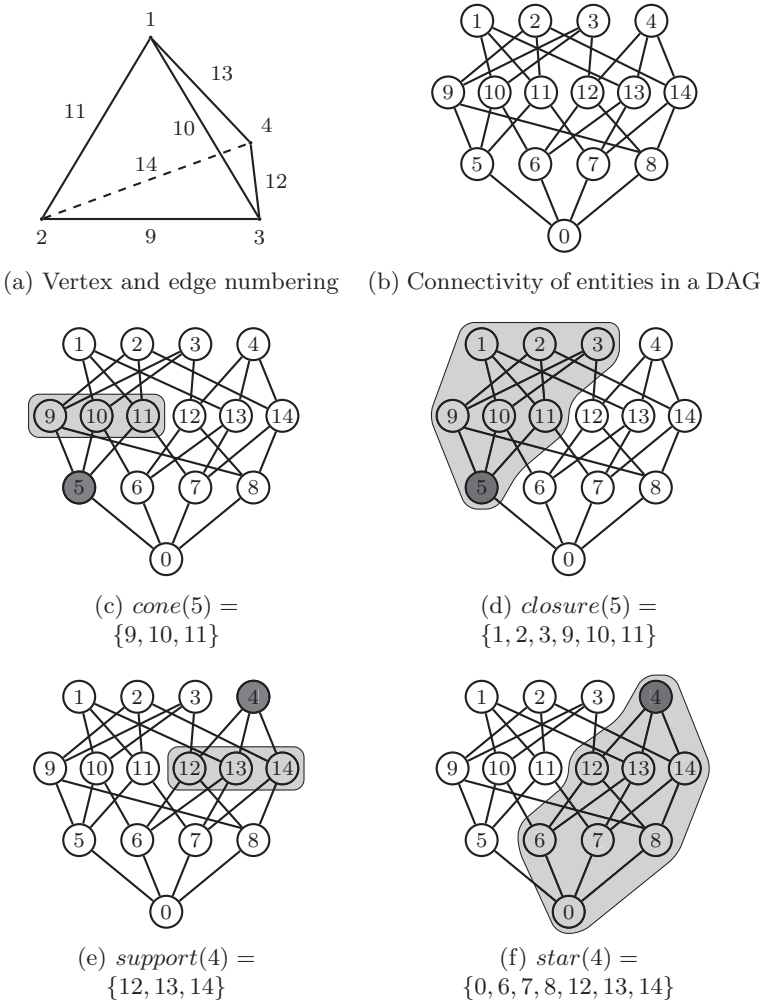
A key component to achieving performance in Firedrake is PyOP2, a high-level framework that optimizes the parallel execution of numerical kernels over unstructured mesh data [24]. PyOP2 represents mesh entities as *sets* and connectivity between them as *mappings*, where input data to the compiled kernel is accessed either directly or indirectly via a *mapping*. In parallel PyOP2 is able to overlap halo data communication with kernel computation during the execution loop due to a specialized data ordering within *sets* [20].

2.2. DMPlex. The ability of PETSc to manage unstructured meshes is centered around DMPlex, a data management object that encapsulates the topology of unstructured grids and provides a wide range of common mesh management functionalities to application programmers [17]. As such, DMPlex provides a domain topology abstraction that decouples user applications from the implementation details of common mesh-related utility tasks, such as file I/O, domain decomposition methods, and parallel load balancing [16], which increases extensibility and improves interoperability between scientific applications through librarization [5].

Similar abstractions for managing unstructured mesh data exist. The PUMI library provides a data model for encapsulating nonmanifold mesh geometries, complete with parallel domain decomposition, data migration capabilities, and predefined discretization definitions [13]. The iMesh component interface [22] defines a generalized data model along with a set of basic capabilities for adaptive mesh generation and manipulation. This has been implemented in MOAB [25] and integrated with mesh generation and adaptation packages [21].

DMPlex uses an abstract representation of the unstructured meshes in memory, where the connectivity of topological entities is stored as a directed acyclic graph (DAG) [15, 18]. The DAG is constructed of clearly defined layers (strata) that enable access to mesh entities by their topological dimension or codimension, enabling application codes to be written without explicit reference to the topological dimension of the mesh. As illustrated in Figure 1(b), all points in the topology DAG share a single consecutive entity numbering, emphasizing that each point is treated equally no matter its shape or dimension, and allowing DMPlex to store the graph connectivity in a single array where dimensional layers are defined as consecutively numbered

subranges. The directional connectivity of the DAG is defined by the covering relationship $cone(p)$, which denotes all points directly connected to p in the next codimension, as illustrated in Figure 1(c). The transitive closure of the *cone* operation is denoted as $closure(p)$ and depicted in Figure 1(d). The dual operation $support(p)$ and its transitive closure $star(p)$ are shown in Figures 1(e) and 1(f), respectively.



(a) Vertex and edge numbering (b) Connectivity of entities in a DAG

(c) $cone(5) = \{9, 10, 11\}$

(d) $closure(5) = \{1, 2, 3, 9, 10, 11\}$

(e) $support(4) = \{12, 13, 14\}$

(f) $star(4) = \{0, 6, 7, 8, 12, 13, 14\}$

FIG. 1. *Example entity numbering for a single tetrahedron and the corresponding internal DAG. Entities are numbered across stratified layers (dimensions) with a consecutive numbering in each stratum.*

In addition to the abstract topology data, PETSc provides two utility objects to describe the parallel data layout: a *section* object maps the graph-based topology information to discretized solution data through an offset mapping very similar to the compressed sparse row (CSR) storage scheme, and the *star forest* (SF) [3] object holds a one-sided description of shared data in parallel. These data layout mappings allow DMPlex to manage distributed solution data by automating the preallocation of distributed vector and matrix data structures and performing halo data exchanges.

Moreover, by storing grid topology alongside discretized solution data, DMPlex is able to provide the mappings required for sophisticated preconditioning algorithms, such as geometric multigrid methods [6] and multiblock, or "Fieldsplit," preconditioning for multiphysics problems [4].

2.3. Mesh reordering techniques. The runtime performance of geometry-based processing algorithms can be significantly affected by the data layout of unstructured meshes and sparse matrices due to caching effects. A number of mesh ordering techniques exist that aim to increase the cache coherency of local data, either through cache-aware or cache-oblivious reordering [26, 11, 12]. Cache-oblivious techniques aim to reduce the bandwidth of the resulting sparse matrix and thus lower the number of cache misses incurred when traversing local data regardless of the underlying caching architecture.

The reverse Cuthill–McKee (RCM) algorithm [7, 9] represents a classic example of a cache-oblivious mesh reordering. RCM is based on a variant of a simplex breadth-first search of the mesh connectivity graph and yields a fixed-size n tuple that represents the new ordering permutation. Alternative methods, such as space filling curve numberings, may be used to create similar permutations from a given mesh topology graph in order to further increase cache coherency.

3. Computational meshes in Firedrake. The Firedrake system comprises a stack of specialized components that implement a set of multilayered abstractions to provide automated finite element computation from a high-level specification [23]. The role of the top-level Firedrake layer is to marshal data between the various subcomponents and to provide the computation layers, PyOP2 and PETSc, with the maps and data objects required to assemble and solve linear and nonlinear systems. The computational mesh is encapsulated in a `Mesh` object that can be either read from file or generated in memory for common geometry classes, such as squares, cubes, or spheres.

A characteristic feature of the Firedrake execution stack is that multiple discretizations of the same computational domain, represented by the `FunctionSpace` class, may be derived dynamically at any point during execution, which requires the topological connectivity of the mesh to be stored in a separate object. Separating mesh topology from the discretization of the problem not only enables Firedrake to exploit caching and data reuse with minimal replication at multiple levels in the tool chain but also allows data layout optimizations to be inherited for all derived discretizations without recomputation of the mesh reordering scheme.

As shown in Figure 2, the Firedrake classes `Mesh` and `FunctionSpace`, which encapsulate mesh topology and problem discretization, respectively, map naturally onto the abstractions provided by the PETSc data management API. The `Mesh` class encapsulates the topological connectivity of the grid by storing a DMPlex object alongside a Firedrake-specific application ordering, while discretization data given by the `FunctionSpace` class defines the layout of local data stored in the `Function` object.

3.1. Mesh topology. Firedrake uses the DMPlex data management abstraction as an internal representation of mesh topology, allowing it to delegate file I/O and runtime mesh generation to PETSc. In doing so, Firedrake depends only on the public API provided by PETSc and automatically inherits the mesh management and manipulation capabilities provided by DMPlex. As a result, Firedrake naturally supports the same set of mesh file formats as DMPlex, which at the time of this writing includes ExodusII, Gmsh, CGNS, MED, and Fluent Case files, and thus increases

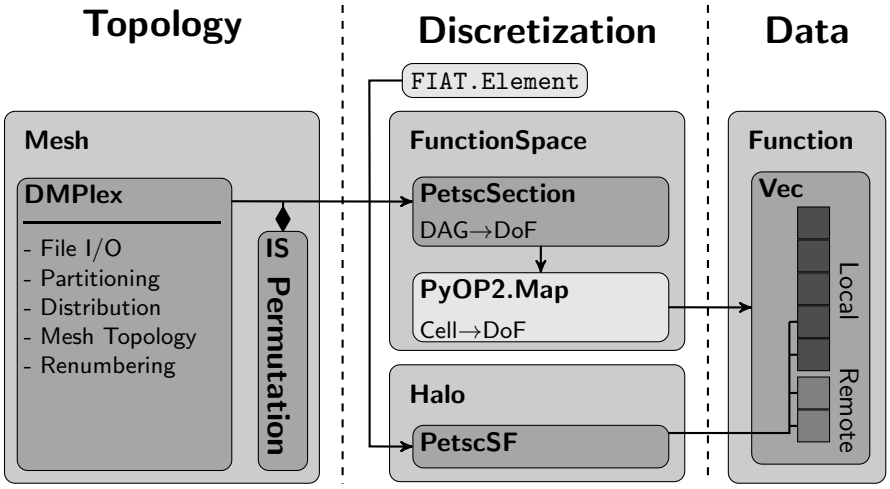


FIG. 2. *Mapping of data abstractions between Firedrake and PETSc: Firedrake's* `Mesh` *object encapsulates domain topology stored in a* `DMPlex` *object alongside an application numbering permutation. The choice of* `FunctionSpace` *defines the local data discretization via a* `PetscSection` *that is used to generate the indirection maps required by PyOP2 for assembly computation. Halo communication is performed by a* `PetscSF` *object, which encapsulates the mapping between local and remote data items in the local* `Vec`.

interoperability with other applications and provides extensibility through a well-supported public library.

In addition to various mesh format readers, DMPlex also provides parallel domain decomposition routines that interface with external libraries, such as Chaco and Metis/ParMetis, to facilitate parallel partitioning of the topology graph. Utilizing the PETSc internal communication routines, DMPlex is thus capable of automatically distributing the mesh across any number of processes, which allows Firedrake to fully automate the parallelization and optimization of the user-defined finite element problem.

Another advantage of using the DMPlex DAG as an intermediate representation of mesh topology is that the abstracted graph format allows Firedrake to dictate the ordering of the mesh topology, and thus control local data layout of derived discretizations. This is made possible by attaching a point permutation to the DMPlex object, which defines a single level of indirection that is applied to all graph traversal operations within DMPlex. As a result, all discretization objects derived from the stored topology inherit this permutation, giving Firedrake an effective way to control the global ordering of derived solution data.

3.2. Discretization. The FEniCS language (UFL [1]) implemented by Firedrake allows the use of various discretization schemes to represent solution data, where the number of DoFs associated with each mesh entity is determined by the local discretization within a reference element. The FIAT package [14] of the FEniCS software stack provides this reference element from which Firedrake needs to derive the indirection *maps* between mesh cells and DoFs required by PyOP2 to perform matrix and vector assembly.

The mapping from mesh topology to solution data is facilitated by PETSc through `PetscSection`, a class of descriptor objects that store a CSR-style mapping between points in the topology DAG and entries in array or vector objects. Assuming a

constant element type throughout the mesh, DMPlex can generate a *section* object, given the number of DoFs associated with each mesh entity type as provided by the FIAT reference element. The set of DoFs associated with a cell can then be derived by taking the *closure* of the cell point (see Figure 1(d)) and collecting the DoFs associated with each closure point by the provided *section*.

The use of DMPlex *closures* to determine entity-to-DoF mappings is sufficient on its own should the local numbering of mesh entities within a cell closure match that required by the application. In Firedrake the local numbering on simplices must match the simplex numbering used in FEniCS [19], where the local facet number is determined by the local number of the opposite vertex. Algorithm 1 is thus applied to each cell closure in turn to enforce the desired local numbering for simplices.

Algorithm 1 Local numbering algorithm for simplex elements.

```
1:  for cell in mesh do                                   ▷ Loop over all cells in the mesh
2:      closure_cell ← DMPlexGetClosure(plex, cell)
3:      for p in closure_cell do                          ▷ Filter facets and vertices from cell closure
4:          if p in DepthStratum(plex, 0) then vertices ← p
5:          if p in HeightStratum(plex, 1) then facets ← p
6:      Sort(vertices)                                    ▷ Sort vertices by global number
7:      for facet in facets do
8:          closure_facet ← DMPlexGetClosure(plex, facet)
9:          for f in closure_facet do                     ▷ Filter vertices from facet closure
10:             if f in DepthStratum(plex, 0) then v_facet ← f
11:         for v in vertices do                          ▷ Find nonadjacent vertices
12:             if v not in v_facet then keys ← (facet, v)
13:     Sort(facets, keys)                                ▷ Sort facets by nonadjacent vertices
```

3.3. Halo communication. The exchange of halo data between processors in Firedrake is performed by the PETSc *star forest* (SF) [3] communication abstraction that encapsulates one-sided description of shared data. SF objects implement a range of sparse communication patterns that are able to perform common data communication patterns, such as broadcasts and reduction operations, over sparse data arrays according to the stored mapping. The halo data exchange pattern is derived by DMPlex from an internal SF encapsulating the overlap in the topology graph and a given discretization provided in the form of a *section* object. The derived SF encapsulates a local-to-local remote data mapping that avoids the need for converting halo data into a global numbering.

4. Application orderings. When deriving function spaces from a DMPlex topology definition, the global data layout is inherited from the original graph ordering generated by PETSc. PyOP2, however, imposes a data layout restriction that allows it to optimize performance by overlapping computation with communication, which is not honored in the global entity numbering generated by DMPlex. Firedrake therefore generates an *application ordering* in the form of a permutation of the DAG points that is passed to a distributed DMPlex object to generate indirection maps that adhere to the ordering required by PyOP2.

4.1. PyOP2 data ordering. To ensure that halo exchange communication can be overlapped with assembly kernel computation, PyOP2 *sets* require a strict entity

ordering, where nonowned data is stored contiguously at the end of the data array. Moreover, as shown in Figure 3, all owned data items adjacent to nonowned items require that the halo data exchange be finished before computation is performed. Thus, owned data is further partitioned into *core* (independent of halo) and *noncore* (halo-dependent) data, allowing processing over *core* data items to proceed while communication is still in-flight.

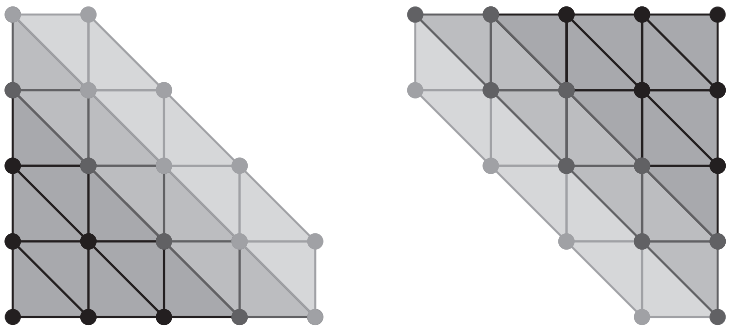


FIG. 3. *PyOP2 entity classes on a distributed* 4×4 *unit square mesh. The dark region marks* core *entities, medium grey marks* noncore *entities, and light grey marks the halo region.*

Firedrake honors the PyOP2 entity ordering by assigning all points in the DMPlex topology DAG to one of the PyOP2 entity classes using a `DMLabel` data structure, which encapsulates integer value assignments to points. When deriving the indirection maps for each discretization, mesh entities can then be filtered into the appropriate sets regardless of entity type. The algorithm used to mark PyOP2 entity classes is shown in Algorithm 2, where the initial overlap definition, provided by DMPlex in the form of an SF, is used to first mark the *halo* region, followed by the derivation of adjacent *noncore* points.

Algorithm 2 Algorithm to mark PyOP2 entity classes on DMPlex based on the initial *halo* definition given by DMPlex. Point adjacency in the DAG is defined as $adjacency(p) = closure(star(p))$.

```
1:  for p in pointSF do                                   ▷ Define halo region from SF
2:      LabelSetValue(halo, p)
3:  for p in LabelGetStratum(halo) do                     ▷ Loop over halo cells
4:      if p in HeightStratum(plex, 0) then
5:          adjacency ← DMPlexGetAdjacency(plex, p)
6:          for c in adjacency do                         ▷ Find cells adjacent to halo
7:              if LabelHasPoint(halo, c) and c in HeightStratum(plex, 0) then
8:                  LabelSetValue(noncore, p)             ▷ Mark adjacent cell as noncore
9:  for p in mesh do                                      ▷ Mark remaining points as core
10:     if not LabelHasPoint(halo, p) and not LabelHasPoint(noncore, p) then
11:         LabelSetValue(core, p)
```

4.2. Compact RCM ordering. The generic encapsulation of mesh topology allows DMPlex to compute the point permutation according to the well-known RCM mesh reordering algorithm (see section 2.3). Since Firedrake already controls the effective ordering of mesh entities to adhere to PyOP2 ordering restrictions, the RCM permutation provided by DMPlex can be applied to the Firedrake-specific point permutation. However, any additional indirection applied to the reordering permutation computed by Firedrake needs to be contained within the marked PyOP2 class regions. Thus, although the base RCM permutation generated by DMPlex includes all graph points, Firedrake implements a cellwise compact reordering, where the cell ordering is filtered from the RCM permutation within each marked PyOP2 region. As shown in Algorithm 3, the full permutation is then derived by adding cell *closures* along the segmented cell order, ensuring the relative compactness of DoFs associated with the same cell.

Algorithm 3 Algorithm for generating a compact RCM permutation that honors PyOP2's entity class separation and encapsulates a cellwise RCM reordering within the PyOP2 regions.

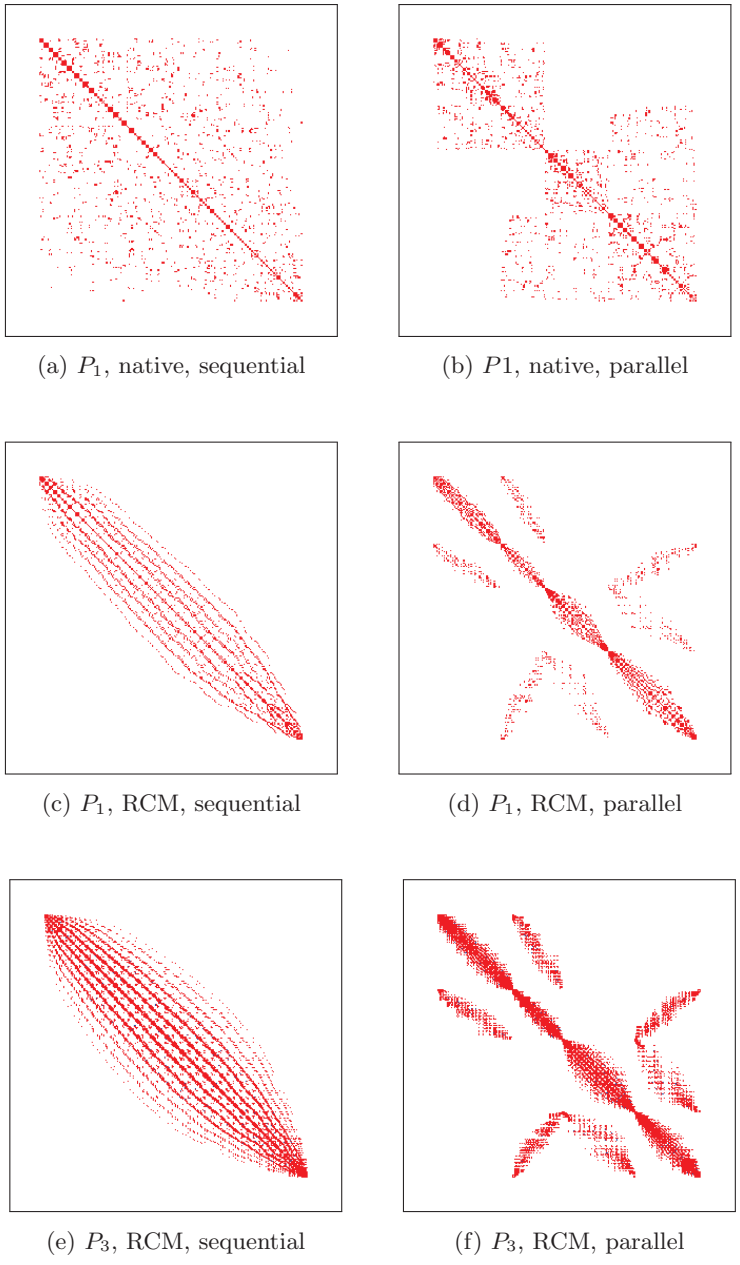
```
1: ordering ← DMPlexGetOrdering(RCM)                    ▷ Get RCM renumbering
2: for class in {core, noncore, halo} do                ▷ Get array index for each class
3:     idx_class ← LabelStratumSize(class)
4: for p in mesh do
5:     p_rcm ← ordering{p}
6:     if p_rcm not in HeightStratum(plex, 0) then skip p
7:     for class in {core, noncore, halo} do            ▷ Get array index for current class
8:         if LabelHasPoint(class, p) then idx ← idx_class
9:         for p_closure in DMPlexGetClosure(plex, p_rcm) do
10:            permutation{idx} ← p_closure
```

It is worth noting that this cellwise compact reordering approach allows any additional level of indirection to be applied without violating the PyOP2 ordering constraint and is therefore not limited to RCM. Examples of sparse matrix structures generated using the RCM-based reordering are shown in Figure 4.

5. Performance benchmarks. The benefits of Firedrake's compact RCM mesh reordering have been evaluated using two sets of performance benchmarks: (1) a runtime comparison of assembly loops over cells and interior facets with lightweight kernels and (2) solving a full advection-diffusion problem. The benchmark experiments were carried out on the UK national supercomputer ARCHER, a Cray XE30 with 4920 nodes connected via an Aries interconnect and a parallel Lustre filesystem.[1] Each node consists of two 2.7 GHz, 12-core Intel E5-2697 v2 (Ivy Bridge) processors with 64 GB of memory.

An indication of the indirection cost and subsequent data traversal performance in low-level loops was gained by comparing the individually measured execution times of two PyOP2 assembly loops. The benchmark loops were generated by invoking `assemble(L)` 100 times for the UFL expressions `L = u*dx` and `L = u('+')*dS` for cell and interior facet integrals, respectively, where `u` is a suitable `Function` object. The performance of a full-scale finite element problem was then analyzed, which consisted

[1] http://www.archer.ac.uk/

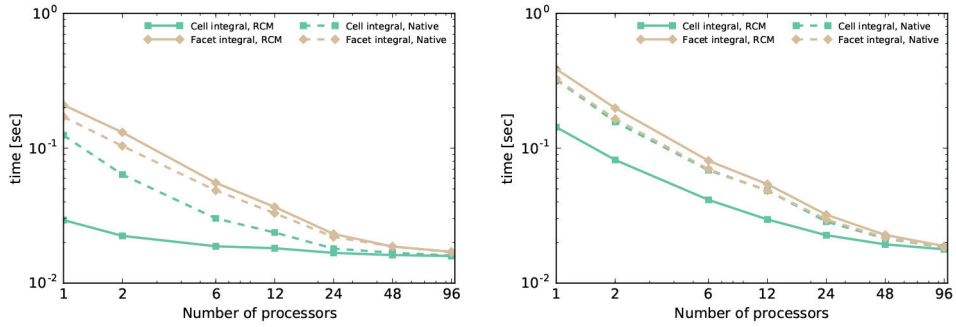


(a) P_1, native, sequential (b) $P1$, native, parallel

(c) P_1, RCM, sequential (d) P_1, RCM, parallel

(e) P_3, RCM, sequential (f) P_3, RCM, parallel

FIG. 4. *Effects of the combined RCM and OP2 mesh ordering on matrix structure for a P_1 and a P_3 function space on a 5×5 unit square.*



(a) PyOP2 loop performance for P_1 (b) PyOP2 loop performance for P_3

FIG. 5. *Runtime comparison between compact RCM and native numbering for assembly loops over cells and interior facets.*

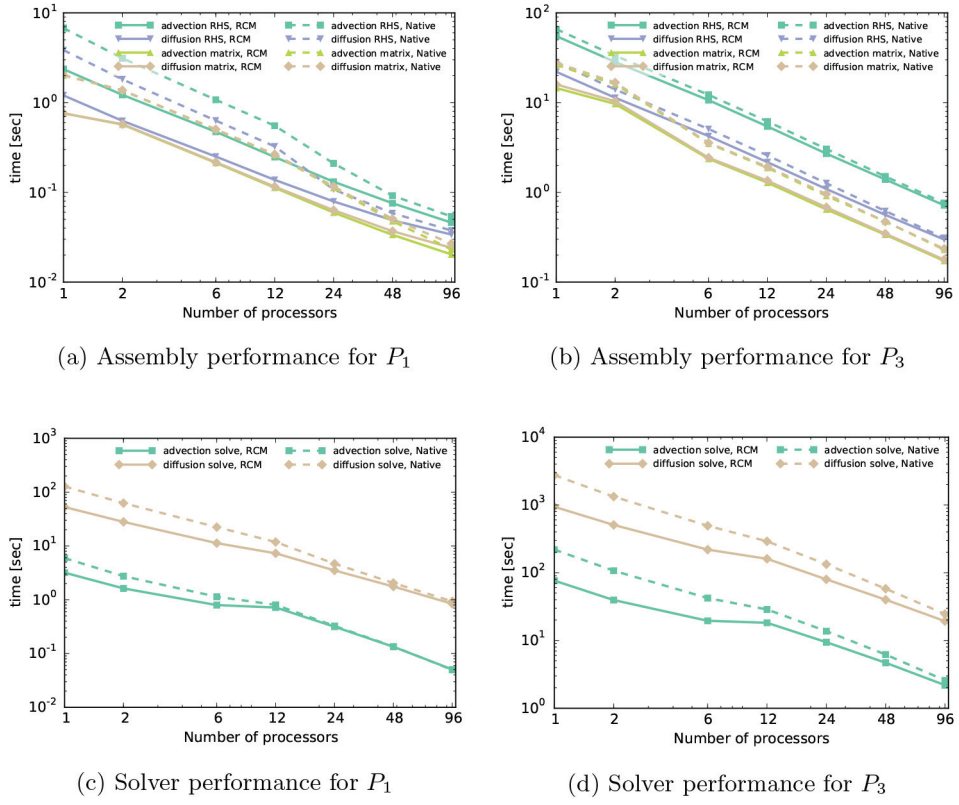
of assembling and solving advection-diffusion equation $\frac{\partial c}{\partial t} + \bigtriangledown\cdot(\vec{u}c) = \bigtriangledown\cdot(\overline{\overline{\kappa}} \bigtriangledown c)$ using the conjugate gradient method with a Jacobi preconditioner for advection and the HYPRE BoomerAMG algebraic multigrid preconditioner [8] for the diffusion component. The mesh used in both experiments represents a two-dimensional L-shaped domain, consisting of 3,105,620 cells and 1,552,808 vertices, and was generated with Gmsh [10].

The performance of the assembly loops over cells and interior facets using P_1 and P_3 function spaces on up to 96 cores is shown in Figure 5. The performance of the cell integral loop shows significant improvements in both cases, whereas the facet integral loop shows a small performance decrease. This highlights that the compact RCM reordering optimizes cell integral computation due to the generated cellwise compact traversal pattern. It is also worth noting that the improvement due to RCM diminishes as we approach the strong scaling limit, although an increase in computational intensity between P_1 and P_3 assembly kernels negates this effect.

A performance profile of the full advection-diffusion model is given in Figure 6. Matrix and RHS assembly times indicate clear performance improvements under compact RCM, with significant speedups for P_1 on small numbers of cores (see Figure 6(a)). As shown in Figure 6(b), P_3 assembly kernels with a higher computational intensity also show significant performance improvements, where matrix assembly in particular benefits from the reordering in a sustained way up to 96 cores. Similarly, advection and diffusion solver times shown in Figures 6(c) and 6(d) indicate a clear speedup on small numbers of cores, while significant improvements are also evident on up to 96 cores for solves with larger numbers of DoFs in P_3.

6. Discussion. In this paper we give a full account of the utilization of the PETSc DMPlex topology abstraction in Firedrake to derive the topological mapping required to solve a wide range of finite element problems. We highlight how the right composition of abstractions can be used to apply well-known data layout optimizations, such as RCM renumbering, to an entire class of problems, and we demonstrate the resulting gains in assembly and solver performance. Our work emphasizes the importance of high-level DSLs and further underlines their potential for achieving performance portability through runtime optimization.

An important corollary of the close integration of DMPlex into the Firedrake framework is the improved interoperability and extensibility of the mesh management



(a) Assembly performance for P_1

(b) Assembly performance for P_3

(c) Solver performance for P_1

(d) Solver performance for P_3

FIG. 6. *Runtime comparison between compact RCM and native numbering for the advection-diffusion problem on P_1 and P_3 discretizations.*

component. Future efforts to improve file I/O and add new meshing capabilities, such as mesh adaptivity, can now be integrated through PETSc DMPlex interfaces. This ensures that computational models built using the Firedrake framework can easily be extended without breaking existing abstractions, and thus enables domain scientists to leverage automated performance optimizations as well as a wide range of simulation features.

REFERENCES

[1] M. S. ALNÆS, A. LOGG, K. B. ØLGAARD, M. E. ROGNES, AND G. N. WELLS, *Unified form language: A domain-specific language for weak formulations of partial differential equations*, ACM Trans. Math. Software, 40 (2014), 9. doi:10.1145/2566630.

[2] S. BALAY, S. ABHYANKAR, M. F. ADAMS, J. BROWN, P. BRUNE, K. BUSCHELMAN, L. DALCIN, V. EIJKHOUT, W. GROPP, D. KAUSHIK, M.KNEPLEY, L. C. MCINNES, K. RUPP, B. SMITH, S. ZAMPINI, AND H. ZHANG, *PETSc Users Manual*, Technical report ANL-95/11 - Revision 3.6, Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL, 2015. Available online at www.mcs.anl.gov/petsc/petsc-3.6/docs/manual.pdf.

[3] J. BROWN, *Star Forests as a Parallel Communication Model*, Technical report, 2011. http://lists.mcs.anl.gov/pipermail/petsc-dev/attachments/20111224/7bfa1e70/attachment.pdf.

[4] J. BROWN, M. G. KNEPLEY, D. A. MAY, L. C. MCINNES, AND B. SMITH, *Composable linear solvers for multiphysics*, in Proceedings of the 11th International Symposium on Parallel and Distributed Computing (ISPDC), IEEE Computer Society, Washington, DC, 2012, pp. 55–62. doi:10.1109/ISPDC.2012.16.

[5] J. Brown, M. G. Knepley, and B. F. Smith, *Run-time extensibility and librarization of simulation software*, IEEE Comput. Sci. Engrg., 17 (2015), pp. 38–45.
[6] P. R. Brune, M. G. Knepley, and L. R. Scott, *Unstructured geometric multigrid in two and three dimensions on complex and graded meshes*, SIAM J. Sci. Comput., 35 (2013), pp. A173–A191. doi:10.1137/110827077.
[7] E. Cuthill and J. McKee, *Reducing the bandwidth of sparse symmetric matrices*, in Proceedings of the 24th National Conference (ACM '69), ACM, New York, 1969, pp. 157–172.
[8] R. D. Falgout, J. E. Jones, and U. M. Yang, *The design and implementation of hypre, a library of parallel high performance preconditioners*, in Numerical Solution of Partial Differential Equations on Parallel Computers, A. M. Bruaset and A. Tveito, eds., Lecture Notes in Comput. Sci. Eng. 51, Springer, Berlin, 2006, pp. 267–294.
[9] A. George and J. W. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall Ser. Comput. Math., Prentice-Hall, Englewood Cliffs, NJ, 1981.
[10] C. Geuzaine and J.-F. Remacle, *Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities*, Internat. J. Numer. Methods Engrg., 79 (2009), pp. 1309–1331. doi:10.1002/nme.2579.
[11] F. Günther, M. Mehl, M. Pögl, and C. Zenger, *A cache-aware algorithm for PDEs on hierarchical data structures based on space-filling curves*, SIAM J. Sci. Comput., 28 (2006), pp. 1634–1650. doi:10.1137/040604078.
[12] G. Haase, M. Liebmann, and G. Plank, *A Hilbert-order multiplication scheme for unstructured sparse matrices*, Inter. J. Parallel Emergent Distrib. Syst., 22 (2007), pp. 213–220.
[13] D. A. Ibanez, E. S. Seol, C. W. Smith, and M. S. Shephard, *PUMI: Parallel unstructured mesh infrastructure*, submitted.
[14] R. C. Kirby, *Algorithm* 839*: Fiat, a new paradigm for computing finite element basis functions*, ACM Trans. Math. Software, 30 (2004), pp. 502–516. doi:10.1145/1039813.1039820.
[15] M. G. Knepley and D. A. Karpeev, *Mesh algorithms for PDE with Sieve* I: *Mesh distribution*, Sci. Program., 17 (2009), pp. 215–230.
[16] M. G. Knepley, M. Lange, and G. J. Gorman, *Unstructured overlapping mesh distribution in parallel*, submitted. Preprint version available online at arXiv:1506.06194[cs.MS].
[17] M. Lange, M. G. Knepley, and G. J. Gorman, *Flexible, scalable mesh and data management using PETSc DMPlex*, in Proceedings of the 3rd International Conference on Exascale Applications and Software (EASC 2015), The University of Edinburgh, Edinburgh, UK, 2015, pp. 71–76. Available online at http://www.easc2015.ed.ac.uk/sites/default/files/attachments/EASC15Proceedings.pdf.
[18] A. Logg, *Efficient representation of computational meshes*, Internat. J. Comput. Sci. Engrg., 4 (2009), pp. 283–295. doi:10.1504/IJCSE.2009.029164.
[19] A. Logg, G. N. Wells, and J. Hake, *DOLFIN: A C++/Python finite element library*, in Automated Solution of Differential Equations by the Finite Element Method, A. Logg, K.-A. Mardal, and G. Wells, eds., Lecture Notes in Comput. Sci. Engrg. 84, Springer, Berlin, 2012, pp. 173–225.
[20] G. R. Mudalige, M. B. Giles, I. Reguly, C. Bertolli, and P. H. J. Kelly, *OP2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures*, in Proceedings of Innovative Parallel Computing (InPar), IEEE Computer Society, Washington, DC, 2012, pp. 1–12.
[21] C. Ollivier-Gooch, *A mesh-database-independent edge-and face-swapping tool*, in Proceedings of the 44th AIAA Aerospace Sciences Meeting and Exhibit, Reno, NV, 2006, AIAA 2006-533.
[22] C. Ollivier-Gooch, L. Diachin, M. S. Shephard, T. Tautges, J. Kraftcheck, V. Leung, X. Luo, and M. Miller, *An interoperable, data-structure-neutral component for mesh query and manipulation*, ACM Trans. Math. Software, 37 (2010), 29. doi:10.1145/1824801.1864430.
[23] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. McRae, G.-T. Bercea, G. R. Markall, and P. H. J. Kelly, *Firedrake: Automating the finite element method by composing abstractions*, submitted. Preprint version available online at arxiv:1501.01809[cs.MS].
[24] F. Rathgeber, G. R. Markall, L. Mitchell, N. Loriant, D. A. Ham, C. Bertolli, and P. H. J. Kelly, *PyOP2: A high-level framework for performance-portable simulations on unstructured meshes*, in Proceedings of High Performance Computing, Networking, Storage and Analysis (2012 SC Companion), IEEE Computer Society, Washington, DC, pp. 1116–1123. doi:10.1109/SC.Companion.2012.134.

[25] T. J. Tautges, R. Meyers, K. Merkley, C. Stimpson, and C. Ernst, *MOAB: A Mesh-Oriented Database*, Technical report SAND2004-1592, Sandia National Laboratories, Albuquerque, NM, 2004.
[26] S.-E. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha, *Cache-oblivious mesh layouts*, ACM Trans. Graph., 24 (2005), pp. 886–893. doi:10.1145/1073204.1073278.